CS 105: Introduction to Computer Science

Prof. Thao Nguyen

Spring 2025

Materials adapted from Dave Wonnacott

Choosing one of Several Possible Algorithms

In most of CS105, we've been happy to just get *one* answer to a problem.

But, sometimes we think of several algorithms, or ways to program an algorithm.

How do we choose? Compare the three things we want from our core software:

- Correctness ... for now, this is a yes-or-no question: we want "yes, it's correct"
  - We'll revisit the question "how do you *know* it's correct?"
  - Upper-level CS explores things like approximation algorithms, "correctness" can be subtle.
- Clarity ... is it possible to write this algorithm in a way that's easily understood?
  - We usually measure this informally, by asking peers who have broad training with many idioms
- Efficiency ... does the code use no more resources than needed?
  - How do we even measure this?
- (Note: for overall software system, *User Interface* can be just as important!)

Measuring Efficiency

What does efficiency mean? We want to ensure that running code minimizes...

- time taken
- frustration created for the user
- energy taken
- carbon footprint

It turns out that the same techniques can be used to understand each/all of these.

Time is the easiest to measure accurately, often the focus of first-year courses.

So ... what does it mean to measure "time taken by an algorithm"?

Time Taken by an Algorithm, Idea #1: *Measure* Time

Python has ways to measure the time used by a program.

(Python time module)

So, can we just use that to measure the time, and give our answer in seconds?

- The code doesn't take any time when it's just sitting there, we need to run it.
- When we run it, we must choose input/parameters ... these affect the time.
- We also pick a computer ... that also affects the time
- So, trying to express our answer in seconds or milliseconds tells us about the execution of the code *for that input*, when run *on that computer*

We need a better idea...

Time Taken by an Algorithm, Idea #2: Time *Function*

If we run code on different inputs, we may find the time taken varies predictably.

Computational complexity: a formula giving work in terms of parameters

How would we *expect* the time needed to vary with *b* and *e* in the following code:

```
def power1(base: float, exp: int) ->
float:
  precondition(exp > 0)
  if exp == 1:
    return base
  else:
    return base*power1(base, exp-1)
```

Time Taken by an Algorithm, Idea #2: Time *Function*

Computational complexity: a formula giving work in terms of parameters

How would we *expect* the time needed to vary with *b* and *e* in the following code:

```
def power2(base: float, exp: int) ->
float:
  precondition(exp > 0)
  result: float = base
  n_exp: int = 1
  while n_exp < exp:
    result = result * base
    n_exp = n_exp+1
  return result
```

Time Taken by an Algorithm, Idea #2: Time *Function*

Computational complexity: a formula giving work in terms of parameters

How would we *expect* the time needed to vary with *b* and *e* in the following code:

```
def power3(base: float, exp: int) -> float:
  precondition(exp > 0)
  if exp == 1:
    return base
  elif exp%2==0:    # if exp is _even_
    half_exp: int = exp//2
    base_to_half_exp: float = power3(base, half_exp)
    return base_to_half_exp * base_to_half_exp
  else:
    return base * power3(base, exp-1)  # this is still true, use it!
```

Things We Might Observe

As we run experiments, some things may have become evident

- speed varies with the amount of work we do (as we've discussed)
- speed also varies with the computer
  - *often*, computers are simply faster or slower than each other, same code is "best"
  - *but*, multi-core algorithms vary a lot more by computer design
- speed also varies with idiom (also language, language implementation)
  - function calls used to be very slow in Python, now, maybe not too bad?
  - finding a "slice" of an array or list may or may not involve a (slow) copying step
  - "tail recursion" is as fast as loops in *some* languages/implementations, slower in others
  - redundant recursive calls can be made fast in *some* research language implementations

Note: Python code usually needs to rely on built-in operations to be really fast,
e.g. try timing the built-in power or sort/sorted algorithm

Finding "Complexity Functions"

Form Hypothesis:

- Measure cost in terms of some maximally-run fixed-time basic operation
  - Warning: remember that some operations are not fixed-time, e.g. [ ] and + for Python lists!
- Return a *tuple* of (result, work-needed-for-it)

Check Hypothesis:

- analytically, i.e., write out postcondition for work-needed-for-it
- experimentally, i.e., run some tests, plot some curves

More about this on Tuesday

Summary

To create efficient software:

- Use built-in library routines unless there's a reason not to
  - e.g., unless you have some sort of special case that can be solved faster
- Find a high-efficiency algorithm
  - take CMSC 106 or 151, and 231 "Discrete Math", and then 340 "Analysis of Algorithms"
  - CMSC 105 fulfills the prerequisite for 106/151 and 231 (231 also relies on high-school algebra)
- Find a set of idioms and a language so that you can express it
  - clearly
  - correctly
  - with elements that your language can handle efficiently